Verilog Review

Lecture 03

Josh Brake

Harvey Mudd College

Outline

- Verilog tips

- Design guidelines and basic idioms

- Bad Verilog Examples/Bug finding

Learning Objectives

By the end of this lecture you should be able to…

- Recall basic Verilog idioms for common digital structures

- Analyze Verilog code to find errors

Verilog Tips

- Think about hardware you want, write code that implies it. Use the idioms and never think about this as coding.

- Watch for warnings in tool. Take them seriously – often a sign of a big problem.

- Block diagrams: draw box with inputs, outputs. Divide into simpler boxes until you can understand it.

Design Guidelines and Idioms for Common Structures

- Simple combinational logic: use assign statements

- For a mux: assign ? :

- Truth tables: always_comb and case statement (default case of x)

- Finite state machines. Three portions: state register, next state logic, output logic.

- Use always_ff, always_comb, not plain always blocks.

- Use logic datatype everywhere except on tristates, and don't use tristates

System Verilog Operators

Listed in order of descending precedence.

Verilog Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
!	logical negation	logical
~	negation	bit-wise
&	reduction AND	reduction
		reduction OR	reduction
~&	reduction NAND	reduction
~|	reduction NOR	reduction
^	reduction XOR	reduction
~^ or ^~	reduction XNOR	reduction
+	unary (sign) plus	arithmetic
-	unary (sign) minus	arithmetic
{ }	concatenation	concatenation
{{ }}	replication	replication
*	multiply	arithmetic
/	divide	arithmetic
%	modulus	arithmetic
+	binary plus	arithmetic
-	binary minus	arithmetic

<<	shift left	shift
>>	shift right	shift
>	greater than	relational
>=	greater than or equal to	relational
<	less than	relational
<=	less than or equal to	relational
==	logical equality	equality
!=	logical inequality	equality
===	case equality	equality
!==	case inequality	equality
&	bit-wise AND	bit-wise
^	bit-wise XOR	bit-wise
^~ or ~^	bit-wise XNOR	bit-wise
		bit-wise OR	bit-wise
&&	logical AND	logical
||	logical OR	logical
?:	conditional	conditional

Harris and Harris, *Digital Design and Computer Architecture*, ARM Ed., p. 185

Bad Verilog: Learning by Counter Example

Bad Verilog Example #1

```verilog
1  module mux(input  logic [3:0] d0, d1,
2             input  logic       s,
3             output logic [3:0] y);
4    always_comb @(posedge s)
5      if (s) y <= d1;
6      else   y <= d0;
7  endmodule
```

Problem: `always_comb` cannot have a sensitivity list.

Bad Verilog Example #2

```verilog
1  module mux2(input  logic [3:0] d0, d1,
2              input  logic       s,
3              output       [3:0] y);
4
5      tristate t0(d0, s, y);
6      tristate t1(d1, s, y);
7  endmodule
```

Problem: No type defined for the output y. First tristate should be activated for ~s.

Bad Verilog Example #3

```verilog
1  module mux3(input  logic [3:0] d0, d1, d2,
2              input  logic [1:0] s,
3              output logic [3:0] y);
4
5    always_comb
6      if (s == 2'b00) y <= d0;
7      else if (s == 2'b01) y <= d1;
8      else if (s == 2'b10) y <= d2;
9  endmodule
```

Problem: No default case. This will imply a latch and thus throw an error (won't compile).
What would happen if you used always instead of `always_comb`?

Bad Verilog Example #4

```verilog
1  module mux8(input  logic [3:0] d0, d1, d2, d3, d4, d5, d6, d7,
2              input  logic [2:0] s,
3              output logic [3:0] y);
4
5    always_comb
6      case (s)
7        1'd0: y <= d0;
8        1'd1: y <= d1;
9        1'd2: y <= d2;
10       1'd3: y <= d3;
11       1'd4: y <= d4;
12       1'd5: y <= d5;
13       1'd6: y <= d6;
14       1'd7: y <= d7;
15       default: y <= 4'bxxxx;
16     endcase
17 endmodule
```

Problem: Inputs for case statement should be 3 bits instead of 1.

Bad Verilog Example #5

```verilog
 1  module and3(input  logic a, b, c,
 2              output logic y);
 3
 4      logic tmp;
 5
 6      always @(a, b, c)
 7      begin
 8          tmp <= a & b;
 9          y <= tmp & c;
10      end
11  endmodule
```

Problem: Missing `tmp` from sensitivity list. Will synthesize properly, but simulate incorrectly.

Bad Verilog Example #6

```
1  module counter(input  logic        clk,
2                 output logic [31:0] q);
3
4    always_ff @(posedge clk)
5      q <= q+1;
6  endmodule
```

Problem: No reset, starts at random value.

Bad Verilog Example #7

```verilog
1  module counter2(input logic clk,
2                  output logic [31:0] q);
3     initial q <= 32'b0;
4
5     always_ff @(posedge clk) q <= q+1;
6  endmodule
```

Problem: initial causes this to work fine in simulation, but starting value is unpredictable in hardware. Can fix with reset.

Bad Verilog Example #8

```verilog
 1  module gates(input  logic [3:0] a, b,
 2               output logic [3:0] y1, y2, y3, y4, y5);
 3
 4      always @(a)
 5          begin
 6              y1 <= a & b; // AND
 7              y2 <= a | b; // OR
 8              y3 <= a ^ b; // XOR
 9              y4 <= ~(a & b); // NAND
10              y5 <= ~(a | b); // NOR
11          end
12  endmodule
```

Problem: Missing b from sensitivity list. Should have used an always_comb statement.

Bad Verilog Example #9

```
 1  module priority_always(input  logic [3:0] a,
 2                         output logic [3:0] y);
 3
 4     // a 4-input priority encoder
 5     always_comb
 6        if      (a[3]) y <= 4'b1000;
 7        else if (a[2]) y <= 4'b0100;
 8        else if (a[1]) y <= 4'b0010;
 9        else if (a[0]) y <= 4'b0001;
10  endmodule
```

Problem: Missing default statement; will infer a latch.

Bad Verilog Example #10

```verilog
module seven_seg_display_decoder(input  logic [3:0] data,
                                 output logic [6:0] segments);
    always_comb
        case (data)
            0: segments <= 7'b000_0000; // ZERO
            1: segments <= 7'b111_1110; // ONE
            2: segments <= 7'b011_0000; // TWO
            3: segments <= 7'b110_1101; // THREE
            4: segments <= 7'b011_0011; // FOUR
            5: segments <= 7'b101_1011; // FIVE
            6: segments <= 7'b101_1111; // SIX
            7: segments <= 7'b111_0000; // SEVEN
            8: segments <= 7'b111_1111; // EIGHT
            9: segments <= 7'b111_1011; // NINE
        endcase
endmodule
```

Problem: No default statement; will infer a latch.

Bad Verilog Example #11

```verilog
1  module latch(input  logic       clk,
2               input  logic [3:0] d,
3               output logic [3:0] q);
4
5     always_latch @(clk)
6        if (clk) q <= d;
7  endmodule
```

Problem: Missing d from sensitivity list.

Bad Verilog Example #12

```verilog
 1  module floprsen(input  logic       clk,
 2                  input  logic       reset,
 3                  input  logic       set,
 4                  input  logic [3:0] d,
 5                  output logic [3:0] q);
 6
 7     always_ff @(posedge clk, posedge reset)
 8        if (reset) q <= 0;
 9        else       q <= d;
10
11     always @(set)
12        if (set) q <= 1;
13  endmodule
```

Problem: Set is in the wrong block. Should put it in the same block as the other signals.

Bad Verilog Example #13

```verilog
1  module twobitflop(input  logic        clk,
2                    input  logic [1:0] d,
3                    output logic [1:0] q);
4
5      always_ff @(posedge clk)
6          q[1] = d[1];
7          q[0] = d[0];
8  endmodule
```

Missing begin and end statement.

Bad Verilog Example #14

```verilog
 1  module FSMbad(input  logic clk,
 2                input  logic a,
 3                output logic out1, out2);
 4
 5      logic  state;
 6
 7      always_ff @(posedge clk)
 8          if (state == 0) begin
 9              if (a) state <= 1;
10          end else begin
11              if (~a) state <= 0;
12          end
13
14      always_comb
15          if (state == 0) out1 <= 1;
16          else            out2 <= 1;
17  endmodule
```

Output logic is missing logic to set the other outputs to 0 in each state.

Bad Verilog Example #15

```
 1  module divideby3counter(input  logic        clk, reset,
 2                          output logic [1:0] q);
 3
 4    always_ff @(posedge clk or posedge reset)
 5      if (reset) q = 0;
 6      else begin
 7        q = q+1;
 8        if (q == 2) q = 0;
 9      end
10  endmodule
```

Problem: Divides by two because it uses blocking assignments. This would work if you used non-blocking assignments.

Bad Verilog Example #16

```verilog
1  module divideby3FSM(input  logic clk,
2                      input  logic reset,
3                      output logic out);
4
5      logic  [1:0] state, nextstate;
6
7      parameter S0 = 2'b00;
8      parameter S1 = 2'b01;
9      parameter S2 = 2'b10;
10
11     // State Register
12     always_ff @(posedge clk, posedge reset)
13         if (reset) state <= S0;
14         else       state <= nextstate;
15 // Next State Logic
16     always_comb
17         case (state)
18             S0: nextstate <= S1;
19             S1: nextstate <= S2;
20             S2: nextstate <= S0;
21         endcase
22
23     // Output Logic
24     assign out = (state == S2);
25 endmodule
```

Problem: Missing default from nextstate logic; infers a latch.

Bad Verilog Example #17

```verilog
 1  module divideby3FSM2(input  logic clk,
 2                       output logic out);
 3
 4      logic  [1:0] state, nextstate;
 5
 6      parameter S0 = 2'b00;
 7      parameter S1 = 2'b01;
 8      parameter S2 = 2'b10;
 9
10      initial state = 2'b00;
11
12      // State Register
13      always_ff @(posedge clk)
14          if        (state == 2'b00) state <= 2'b01;
15          else if (state == 2'b01) state <= 2'b10;
16          else if (state == 2'b10) state <= 2'b00;
17
18      // Output Logic
19      assign out = (state == S2);
20  endmodule
```

Works in sim, could get stuck in state 2'b11 in hardware. Can fix with a default or a reset.

Bad Verilog Example #18

```verilog
 1  module adventuregameFSM(input              clk, reset, N, S, E, W,
 2                          output logic [3:0] room,
 3                          output logic       win, die);
 4
 5    always_ff @(posedge clk or posedge reset)
 6      if (reset) begin
 7        room <= 4'b0001;
 8        die <= 0;
 9        win <= 0;
10      end
11
12      else case (room)
13        4'b0001: if (E) room <= 4'b0010;
14                 else die <= 1;
15        4'b0010: if (S) room <= 4'b0100;
16                 if (W) room <= 4'b0001;
17                 else die <= 1;
18        4'b0100: if (E) room <= 4'b1000;
19                 if (N) room <= 4'b0010;
20                 else die <= 1;
21      endcase
22  always_comb
23      if (room == 4'b1000) win <= 1;
24  endmodule
```

Win is assigned in two different always blocks. Missing elseifs in two blocks. Missing default cases.

Wrap Up

- Think about hardware you want. Then write the HDL to imply the proper logic.

- Check the Netlist Analyzer to ensure that the tool is inferring the logic you are intending.

- Make sure to not infer latches.

Announcements and Reminders